15-451/651 Algorithms, Fall 2021
Recitation #4 Worksheet

Recap of this week's lectures:

- Picking a random prime, and the density of primes
- String equality testing and Karp-Rabin Fingerprinting
- Dynamic Programming

Fingerprinting

A String Matching Oracle: In this recitation we generalize the fingerprinting method described in lecture. Let $T = t_0, t_1, \ldots, t_{n-1}$, be a string over some alphabet $\Sigma = \{0, 1, \ldots, z-1\}$. Let $T_{i,j}$ denote the substring $t_i, t_{i+1}, \ldots, t_{j-1}$. This string is of length $j - i$. We want to preprocess T such that the following comparison of two substrings of T of length ℓ can be answered (with a low probability of a false positive) in constant time:

$$\text{Test if } T_{i,i+\ell} = T_{j,j+\ell}$$

First of all let's define the fingerprinting function. Let p be a prime, along with a base b (larger than the alphabet size). The Karp-Rabin fingerprint of T is

$$h(T) = (t_0 b^{n-1} + t_1 b^{n-2} + \cdots + t_{n-1} b^0) \bmod p$$

From now on we will omit the mod p from these expressions.

Now, to preprocess the string T, we will compute the following arrays for $0 \leq i \leq n$:
(*Don't forget we are omitting the* mod*s!*)

$$\begin{aligned} r[i] &= b^i \\ a[i] &= t_0 b^{i-1} + t_1 b^{i-2} + \cdots + t_{i-1} b^0 \end{aligned}$$

Give algorithms for computing these in time $O(n)$:

Solution:

$$r[i] = \begin{cases} 1 & \text{if } i = 0 \\ r[i-1] * b & \text{otherwise} \end{cases}$$

$$a[i] = \begin{cases} 0 & \text{if } i = 0 \\ a[i-1] * b + t_{i-1} & \text{otherwise} \end{cases}$$

Notice that

$$h(T_{i,j}) = a[j] - a[i] \cdot r[j-i]$$

Prove that this is the correct expression.

Solution:

$$\begin{array}{rcccc} a[j] & = & t_0 b^{j-1} + t_1 b^{j-2} + \cdots + t_{i-1} b^{j-i} & + & t_i b^{j-i-1} + \cdots + t_{j-1} b^0 \\ & = & a[i] \cdot b^{j-i} & + & h(T_{i,j}) \end{array}$$

So the end result is that we can test if $T_{i,i+\ell} = T_{j,j+\ell}$ by comparing $h(T_{i,i+\ell})$ with $h(T_{j,j+\ell})$. The probability of a false positive can be made as small as desired by picking a sufficiently large random prime p, as seen in lecture. (Here we are not concerned with bounding the false positive probability.)

Extension to string comparison: Suppose we want to know not just if $T_{i,i+\ell}$ equals $T_{j,j+\ell}$, but we want to know the result of comparing these two strings. (That is we want to know if the first is less, equal to, or greater than the second.)

Give an algorithm to do this that runs in $O(\log \ell)$ time:

Solution: Do a binary search to find the smallest k in $0 \le k \le \ell$ where $T_{i,i+k}$ and $T_{j,j+k}$ differ. Return the result of the comparison $t_{i+k} : t_{j+k}$.

Implementation notes:

- When implementing these algorithms it's important to understand the difference between the mathematical mod operator and the "`%`", or the "`mod`" operator that appears in many programming languages. Usually (e.g. in C, C++, Java, Ocaml,...) the value of "`a % b`" has the same sign as `a`. e.g. `(-3) % 5` is `-3`, but using the mathematical mod , as we are using in these notes, $(-3) \bmod 5 = 2$. You must take this into account, or your code will not work.

- The mod operator must be applied often enough so as to guarantee that no overflow occurs. For example if you're computing $\left(\sum_{i=0}^{1000} f_i * g_i\right) \bmod p$. Say you're working in 64-bit signed arithmetic. If the f_i and g_i are up to, say, 10^9, then this sum will probably overflow the available 64 bits. So what you have to do is take the mod after adding each $f_i * g_i$ term into the summation.

- The danger of false positives increases with the increased number of tests being done. So be aware that this must be considered in any deployment of these algorithms. One way to mitigate the effect of false positives is to compute the hash function two or more times using different primes in place of p, or different base values in place of b. Another way is to judiciously check tests that return "equal" to guarantee that the overall algorithm is computing the correct result. (I am not aware of how to do this efficiently in the case of the comparison algorithm described here.)

Palindromes: Given an alphabet Σ, call a string $s \in \Sigma^*$ with $|s| \geq 1$ an *almost-palindrome* if there exists some $s' \in \Sigma^*$ with $|s| = |s'|$ such that s and s' differ in at most one character. For example, "bad" is an almost-palindrome because it is a palindrome if we change the b to a d and "racecar" is an almost-palindrome because it is itself a palindrome. However, "abc" is not an almost-palindrome because it would take at least two character changes to make it into a palindrome.

Given a text $T \in \Sigma^n$ represented as an array of characters, devise an algorithm to count the number of substrings of T that are almost-palindromes in $O(n \log n)$ time. (Hint: Karp-Rabin fingerprinting!)

Solution: Using the fingerprinting approach discussed in the previous question, we can perform an $O(n)$ preprocessing step (namely, computing the rolling hash) after which we may obtain a hash value for $T_{i,j}$ in $O(1)$ time. Moreover, note that the same holds for the reversed text (let's denote this T^R). This means that given any indices $0 \leq i \leq j < n$, we can obtain hash values for both $T_{i,j}$ and $(T_{i,j})^R = T^R_{n-1-j,n-1-i}$.

Let us first consider odd-length almost-palindromes. Observe that every such palindrome has a center character. Our approach, then, is to count the number of almost-palindromes with centers at each index. Given that the palindrome is centered at some index i, we proceed by conducting a binary search using our hash values to find the least k such that $T_{i-k,i}$ does not equal $(T_{i,i+k})^R$. Then we can say that $T[i-k]$ will be the (at most one) error in our almost-palindromes centered at i, since we can set $T[i-k] = T[i+k]$ to maintain the palindrome property. Finally, we do another binary search to find the greatest j such that $T_{i-k-1-j,i-k-1}$ equals $(T_{i+k+1,i+k+1+j})^R$. Then there are $k+j+1$ almost-palindromes centered at index i; the first k have no errors, and the subsequent $j+1$ have only $T[i-k] \neq T[i+k]$.

The idea is similar for even-length palindromes; we leave it as an exercise to work out the details.

As every palindrome has a distinct center, we can perform this for each of the n possible centers of the palindrome and add the results. The time complexity of the entire algorithm is then n centers times $O(\log n)$ cost for each center, plus the one-time $O(n)$ cost to compute the a and r arrays necessary for arbitrary $T_{i,j}$ hashes, which is $O(n \log n)$ in total.

Note that the error is almost one-sided. i.e. if a string is not an almost palindrome, the algorithm may count it as a palindrome due to collisions. It a string is a palindrome with 1 error, the algorithm may take it as a palindrome, and count it as a substring of an almost palindrome. However, it's not possible to completely miss the pattern in the count. If it is given that the length and number of almost palindromes is much less than n (i.e. the pattern is sparse), we could convert the Monte Carlo algorithm (incorrect with small probability) to a Las Vegas algorithm (guaranteed correctness with randomized runtime). At the end of each iteration, we check verify each almost-palindrome that the algorithm finds, and terminates the program only if all the almost-palindromes are correct. The expected number of iterations it takes to get a correct count is constant (Use geometric random variables).

Dynamic Programming

Matrix: You are given a positive integer N and a 26 by 26 matrix A whose entries are either 0 or 1. How many strings of length N, consisting of lowercase English letters, are there such that for all $1 \leq i, j \leq 26$ where $A_{ij} = 0$, character j does not appear directly after character i? Give a dynamic programming solution that runs in $O(N)$ time.

Solution:

Let $DP[n][c]$ be the number of valid strings of length n, ending with character c. Then $DP[1][c] = 1$ for all c, and for all $n > 1$, for all c,

$$DP[n][c] = \sum_{k, A[k][c]=1} DP[n-1][k]$$

Here we are casing on the $(n-1)$th character of the string. Finally the answer is the sum of the values in the row $DP[N]$. There are $26N$ DP states, each of which takes constant time to calculate.

Bonus: Modify the above algorithm to achieve a complexity of $O(\log N)$ time.

Solution: Imagine that the entire DP table has been calculated as in part (a), and imagine each row of the DP table as a vector of length 26. For example, $DP[1]$ is a vector of 26 ones. Observe that for all n, each entry in $DP[n]$ is a linear combination of values in $DP[n-1]$. Also the transition rules do not change as n increases; $DP[i][c]$ is a linear combination of elements in $DP[i-1]$, and $DP[i+100][c]$ is the exact same linear combination of elements in $DP[i+99]$! Thus intuitively there should be a matrix M such that for all n, $DP[n-1] \cdot M = DP[n]$.

So fix an arbitrary n. We want to find a 26 by 26 matrix M such that $DP[n-1] \cdot M = DP[n]$. For all c, by the definition of matrix multiplication, $DP[n][c] = \sum_{k=1}^{26} DP[i-1][k]M[k][c]$. So if $M = A$, we get $DP[n][c] = \sum_{k=1, A[k][c]=1}^{26} DP[n-1][k]$, which exactly matches the definition of $DP[n][c]$! Since c was arbitrarily chosen, A is the matrix we were looking for.

Thus for all n, $DP[n-1] \cdot A = DP[n]$. We can easily get the answer from $DP[N]$, and $DP[N] = DP[1] \cdot A^{N-1}$. So using fast matrix exponentiation, we can compute $DP[N]$,

without computing all the intermediate rows 2 through $N-1$ in the DP table. This takes $O(26^3 \log N) \subseteq O(\log N)$ time using naive matrix multiplication.

Bin-Packing: You are given a collection of n items, and each item has size $s_i \in [0, 1]$. You have many bins, each of unit size, and you want to pack the n items into as few bins as possible. (Each bin can take a subset of items, whose total size is at most 1.)

Show that you can solve this problem in time $O(4^n)$. Then improve this bound to $O(3^n)$.

Solution:

Define $DP(X)$ to be the minimum number of bins needed to pack X, an arbitrary non-empty subset of the items. Here is the recurrence for $DP()$:

$$DP(X) = \begin{cases} 1 & \text{if } X \text{ can be packed into one bin} \\ \min_{Y \subset X,\, Y \neq X,\, Y \neq \emptyset} DP(Y) + DP(X \setminus Y) & \text{otherwise} \end{cases}$$

The runtime is $O(3^n)$, because $\sum_{A \subseteq [1,2,\ldots,n]} \sum_{B \subseteq A} 1 = 3^n$. You can show this by bijecting iterations of the inner loop to base 3 strings of length n. The bijection represents each element in $[1, 2, \ldots, n]$ as a digit in the string. For each element, its digit is 2 if it's in B, 1 if it's in $A \setminus B$, and 0 otherwise.

In practice, to loop over all nonempty subsets of a subset A, consider

```
for(int B = A; B > 0; B = (B - 1) & A)
```

Bonus: solve bin packing in $O(n2^n)$ time.

Solution:

For all subsets S of the items, define $DP(S)$ as the minimum number of bins to hold S, and $R(S)$ to be, out of all ways to pack S into $DP(S)$ bins, the maximum possible remaining space in a single bin. Now clearly

$$DP(S) = \min_{i \in S} \left(DP(S \setminus i) + \begin{cases} 0 & s_i \leq R(S \setminus i) \\ 1 & \text{otherwise} \end{cases} \right)$$

and you can convince yourself that

$$R(S) = \max_{i \in S} \left(\begin{cases} R(S \setminus i) - s_i & s_i \leq R(S \setminus i) \\ 1 - s_i & \text{otherwise} \end{cases} \right)$$

where i ranges only over i that were optimal in the computation of $DP(S)$.

Separate explanation by Danny Sleator:

There's the standard subset DP for this problem that runs in $O(3^n)$ However there's an $O(n2^n)$ algorithm, explained here.

What you do is keep track of two things for a given subset s of weights: (1) the minimum number of bins possible for this subset $b(s)$, and (2) among all solutions that use $b(s)$ bins keep how much stuff is in the minimally used bin, call it $u(s)$.

Now consider the sets in topological (increasing order). To process a set s, consider each of its elements e one at a time. Remove e from s and look this up in the table. Try to fit e into the most-empty bin. If it does fit, put it there. If it does not fit then start a new bin. Keep the best values of $b(s)$ and $u(s)$ over all e.

Naively you might think that the $u()$ values would be computed incorrectly because that bin you put e into might not be the least filled bin among all bins in that solution. But if that is the case then there is some other ordering of insertion where the least filled bin IS last. This will be considered when some other element of s is considered.

Okay, maybe this way of looking at it is more convincing. Consider the optimal solution for set s. Order the bins such that the very last bin is the one which is least full. Assume for the moment that this last bin has more than one item in it, and call one of the items x. Now if we remove x from s, to create s', and look this solution in our DP table, it can be no better than the one just produced by removing x from the solution of s. This is because if there were another solution for s' with more empty space in its most empty bin we could put x in that one and create a better solution for s. The same argument shows that s' cannot use fewer bins than s. (The case when the most empty bin for s contains one element is easy to work out in this context.)

This algorithm is implicitly considering all of the $n!$ orderings with which you could pack the items into bins. But it does this in $O(n2^n)$ instead of $O(n!)$.